

(Edited February 10, 2022)

Introduction

In this CAD you will design the program counter (PC) for your microprocessor. You will be introduced to industry-standard CAD tools for synthesis and APR (automatic place & route).

Program Counter

All instructions start by using the contents of the program counter (PC) as the address to fetch the next instruction. The PC stores the current instruction address and calculates the address of the next sequential instruction. Storing the current instruction address requires nothing more than a resettable 16-bit register. Except on jumps and branches, the next instruction follows sequentially from the current instruction (ie: $PC \leftarrow PC + 1$). On branches and jumps, you should take the new PC address as an input from the datapath. Additionally, the PC needs to be able to stall (keep the same value) for instruction cache misses. When designing deeper pipelines, you may want to compute the branch target address in the PC to save one cycle delay for branches, but this is optional for our 2-stage baseline processor. Table 1 lists the instructions from the baseline architecture that may force an address different from the next sequential address in the PC.

Instruction	Next PC Value
Bcond disp	$PC \leftarrow PC + \text{disp (sign ext.)}$ or $PC \leftarrow PC + 1$
Jcond Rdest	$PC \leftarrow R\text{dest}$ or $PC \leftarrow PC + 1$
JAL Rlink, Rdest	$R\text{link} \leftarrow PC + 1$ and $PC \leftarrow R\text{dest}$

Table 1: PC Modifying Instructions

To avoid confusion concerning the value of the displacement for branches, consider the output listing below from the assembler. Note that the displacement for the ble at 0x141 is 0x0F (e.g. $0x142 + 0xF = 0x151$), not 0x10! Defining the displacement in this manner makes the design of your program counter simpler since the PC has been incremented to 0x142 by the time you're ready to calculate the new branch address.

A
arises due to
the

```

. 0140 / 0020; #0000000000100000 ( 99) test5 or r0 r0
. 0141 / C70F; #1100011100001111 (100) ble j1
. 0142 / 0434; #0000010000110100 (101) xor r4 r4
.          #0000000101010001 (103) .orig 0x0151
. 0151 / 0020; #0000000000100000 (104) j1 or r0 r0
. 0152 / CC0F; #1100110000001111 (105) blt j2
. 0153 / 0434; #0000010000110100 (106) xor r4 r4

```

complication
the fact that
processor is
pipelined. One must decide how to handle changes in program flow in general. In our case, the change in program flow occurs in the second stage. Meanwhile, unless we design the control differently, the processor will continue to fetch and decode subsequent instructions that follow the branch or jump. Several possible solutions are:

1. Always follow a branch or jump instruction with a NOP, or with other instructions that should be executed anyway. This is done by the compiler in many RISC processors and is the simplest solution from a hardware point of view (think about it – you are essentially fixing the problem wholly in software). Often, these branch delay slots can be filled with useful instructions.
2. Lock the first stages of the pipeline so that they do not continue to fetch and decode instructions following a jump or branch. This is equivalent to forcing NOPs in the hardware. This approach adds some complexity, while reducing code size somewhat, and reducing throughput somewhat.
3. For conditional branches, guess that the branch will not be taken and continue fetching and decoding instructions; then squash mis-predicted instructions. This is the most rudimentary type of branch prediction and is only slightly more complicated than (2) but it loses less throughput with the same code size advantage.
4. For conditional branches, guess whether the branch will be taken or not and squash if wrong. This is significantly more complicated, and it requires a fast decode (to know whether the instruction is a conditional branch or jump), a separate arithmetic unit to calculate PC values, and everything required for (3), but it has the potential of being the fastest.

Option 1 is acceptable for the baseline machine.

So, what does a PC need to do? You should be able to initialize the register to a known state, though this state may not necessarily be x0000, depending on your application. For example, sometimes processors have operating modes such that they come out of reset fetching from internal memory in one mode and from external memory in another mode. It must also increment, load in a value for jumps and either load in a value from the ALU or calculate and load a target address for branches. Your program counter must also be made scannable in an effort to improve the testability of your design. You can implement a simple single scan chain. The scan means that your PC must have another mode where it acts as a shift register with one-bit shift in and a one-bit shift out.

Procedure

Register Implementation

From Table 1 it is clear that the next stored PC value could come from a variety of places: it could come from a counter that increments by 1 (normal), the register file (Jump), or an adder (Branch). The PC-unit block shown in your baseline architecture block diagram is assumed to include an adder. It would also be possible to use the ALU to increment in the normal case too, thereby reducing chip area, but complicating control and routing, and lengthening the critical path. The method implied by the bus connections in the baseline architecture block diagram is shown below in Figure 1.

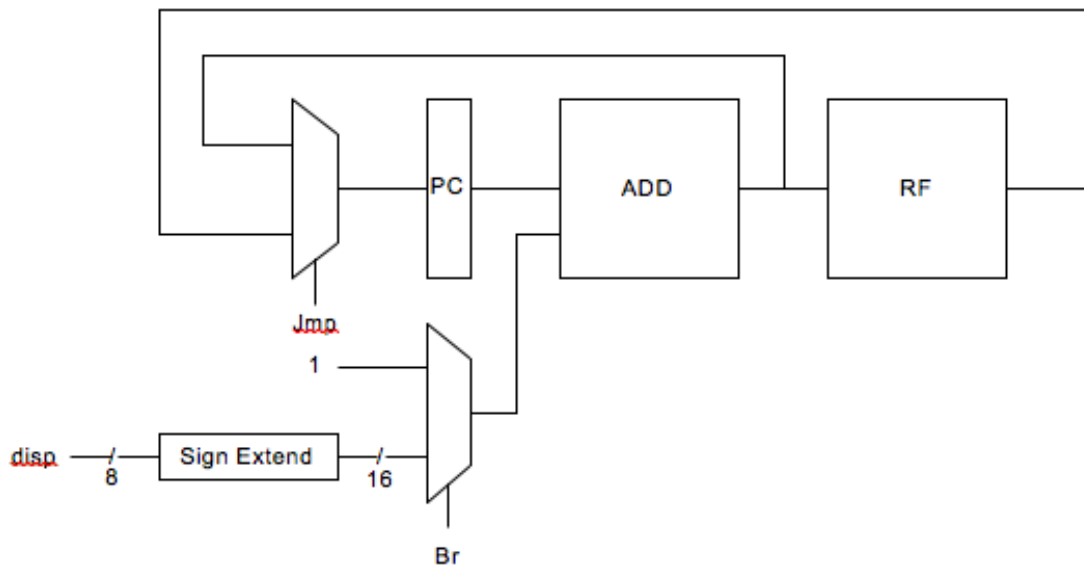


Figure 1: One method of PC Implementation

Synchronous Counter Implementation

You may also design the PC/"incrementor" as a 16-bit synchronous up-counter with reset and parallel loading capability. In the common case, it will auto-increment every clock cycle. The parallel loading ability would be used for loading the branch/jump target calculated by the ALU.

Counters are one of the simplest types of sequential circuits. A counter is usually constructed from two or more flip-flops which change state in a prescribed sequence when input pulses are received. Synchronous counters have an advantage over asynchronous counters in that the stages are clocked simultaneously, and the outputs change in a synchronous manner.

Comments & Requirements

- Go through Tutorial 3 on your own to get some familiarity with the two EDA (stands for “electronic design automation”) tools you will be using: Synopsys Design Compiler and Cadence Innovus. Ultimately you can modify the tutorial files so that they work for your CAD6 design. These are new tools and you may find them confusing to use/understand. Please go to Office Hours to get help if needed.
- Your PC circuitry must properly execute the instructions in the baseline architecture. Be sure that it will function properly in the pipelined environment. If you add interrupt capability to your processor, you must make appropriate extensions to the PC unit. Be sure you can easily reset your PC, or set it to some predetermined state.
- The PC contents are important state information, which you will want to be able to control and observe when testing your processor. Therefore, these should all be implemented in scannable flip-flops or latches. Scan is a concept relevant to DFT, which is covered in discussion. Use this opportunity to gain some exposure to DFT.
- The Verilog model for the off-chip instruction ROM is provided, and you can find it in ROVSD16x1024/Inst_rom_gx (functional view). It may be a good idea to read the spec sheet for this ROM, since this is the block that your PC will be driving. The spec of ROVSD16x1024 is located in the ibm13/resource/ folder. The Innovus manual is also located there.

FAQ

Q: Instead of Verilog can I use SystemVerilog?

A: Yes. It just requires minor changes to your synthesis TCL script. You can use this line to include your SystemVerilog files: `read_file -f sverilog [list “”]`

Q: Help! Something is weird with my design after importing the layout and schematic into Virutoso. DRC/LVS is failing, or NCVerilog is broken!

A: Look in your synthesis log for **Warnings** after the command “check_design” is run. These warnings are seemingly innocuous in Design Compiler and Innovus, but they may be incompatible with NCVerilog.

Submission

Please refer to the submission guidelines mentioned in the CAD1 assignment, as they apply to all CAD assignments. Do not lose points over trivial mistakes like incorrect file names, etc!

For CAD6, you need to submit the following:

- PC:
 - Behavioral Verilog for the PC (make your code clean & easy to understand)

- Design Compiler scripts for the PC (no latches after synthesis)
- Innovus APR scripts for the PC
- Verilog simulation waveforms for both behavioral Verilog and post-APR netlist (need to include the instruction ROM – see the IMEM tutorial that was posted).
- Layouts/Schematic for the PC.
- Clean DRC & LVS reports.
- README:
 - Worst case delay number.
 - Worst case delay path.
 - Describe how you decided your timing constraint.
 - Describe how you decided your floorplan ratio.
 - Describe the approaches you used to implement the PC and why you used those approaches.
 - Describe how branch targets are calculated (in PC or in ALU).
 - Other comments.

Deadline

You must complete CAD6 by **Feb. 24, 2022** by **11pm**.
Do not modify any files in your CAD directory after that time.